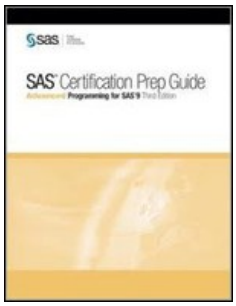

SAS Certification Prep Guide: Advanced Programming for SAS 9, Third Edition

by SAS Institute

Chapter 5: Creating and Managing Tables Using PROC SQL

Overview

Introduction

By using PROC SQL, you can create, modify, and drop (delete) tables quickly and efficiently. Many PROC SQL statements are quite versatile, enabling you to perform the same action in several ways. For example, there are three methods of creating a table by using the CREATE TABLE statement:

- creating an empty table (a table without rows) by defining columns
- creating an empty table that has the same columns and attributes as another table
- creating a table from a query result.

The following PROC SQL step uses the CREATE TABLE statement to create an empty table by defining columns, and uses the DESCRIBE TABLE statement to display information about the table's structure in the SAS log:

```
proc sql;
   create table work.discount
          (Destination char(3),
          BeginDate num Format=date9.,
          EndDate num format=date9.,
          Discount num);
   describe table work.discount;
```

Table 5.1: SAS Log

```
1   proc sql;
2      create table work.discount
3             (Destination char(3),
4             BeginDate num Format=date9.,
5             EndDate num format=date9.,
6             Discount num);
NOTE: Table WORK.DISCOUNT created, with 0 rows and 4 columns.
7       describe table work.discount;
NOTE: SQL table WORK.DISCOUNT was created like:
*

create table WORK.DISCOUNT(bufsize=4096)
            (
            Destination char(3),
            BeginDate num format=DATE9.,
            EndDate num format=DATE9.,
            Discount num
            );
```

* For more information about the BUFSIZE= option, see "Using the BUFSIZE= Option" on page 715.

In this chapter, you will learn to create and manage tables by using the PROC SQL statements shown above, and many others.

Objectives

In this chapter, you learn to

- create a table that has no rows by specifying columns and values
- create a table that has no rows by copying the columns and attributes from an existing table
- create a table that has rows, based on a query result
- display the structure of a table in the SAS log

- insert rows into a table by listing values
- insert rows into a table by specifying column-value pairs
- insert rows into a table, based on a query result
- create a table that has integrity constraints
- set the UNDOPOLICY option to control how PROC SQL handles errors in table insertions and updates
- display the integrity constraints of a table in the SAS log
- update values in existing rows of a table by using one expression and by using conditional processing with multiple expressions
- delete rows in a table
- add, modify, or drop (delete) columns in a table
- drop (delete) entire tables.

Prerequisites

Before beginning this chapter, you should complete the following chapters:

- "Performing Queries Using PROC SQL" on page 4
- "Performing Advanced Queries Using PROC SQL" on page 29
- "Combining Tables Horizontally Using PROC SQL" on page 86.

Understanding Methods of Creating Tables

You can use PROC SQL to create a table in three ways. The *CREATE TABLE* statement is used for all three methods, although the syntax of the statement varies for each method.

Method of Creating a Table	Example

create an *empty table* by *defining columns*	`proc sql;`
	`  create table work.discount`
	`  (Destination char(3),`
	`  BeginDate num Format=date9.,`
	`  EndDate num format=date9.,`
	`  Discount num);`

create an *empty table* that is *like* (has the same columns and attributes as) an existing table	`proc sql;`
	`  create table work.flightdelays2`
	`  like sasuser.flightdelays;`

create a *populated table* (a table with both columns and rows of data) *from a quay result*	`proc sql;`
	`  create table work.ticketagents as`
	`  select lastname, firstname,`
	`  jobcode, salary`
	`  from sasuser.payrollmaster,`
	`  sasuser.staffmaster`
	`  where payrollmaster.empid`
	`  = staffmaster.empid`
	`  and jobcode contains 'TA';`

The CREATE TABLE statement generates only a table as output, not a report. The SAS log displays a message that indicates that the table has been created, and the number of rows and columns it contains.

Table 5.2: SAS Log

```
NOTE: Table WORK.FLIGHTDELAYS2 created, with 0 rows and 8 columns.
```

Note You can display additional information about a table's structure in the SAS log by using the DESCRIBE TABLE statement in PROC SQL. The DESCRIBE TABLE statement is discussed later in this chapter.

Creating an Empty Table by Defining Columns

Overview

Sometimes you want to create a new table that is unlike any existing table. In this case, you need to define all of the columns and attributes that the table will have. To accomplish this, use the *CREATE TABLE statement* that includes *column specifications* for the columns that you want to include. This statement creates a table without rows (an empty table).

Note In addition, integrity constraints can be specified in the CREATE TABLE statement. Integrity constraints are discussed later in this chapter.

General form, basic CREATE TABLE statement with column specifications:

```
CREATE TABLE table-name
       (column-specification-1<,
       …column-specification-n>);
```

where

table-name

specifies the name of the table to be created.

column-specification

specifies a column to be included in the table, and consists of

column-definition <column-constraint-1<, …column-constraint-n>>

<MESSAGE='*message-string*'<MSGTYYE=*message-type*>>

where

column-definition consists of the following:

column-name data-type<(column-width)><column-modifier-1< …column-modifier-n>>

column-name

specifies the name of the column. The column name is stored in the table in the same case that is used in *column-name*.

data-type

is enclosed in parentheses and specifies one of the following: CHARACTER (or CHAR) | VARCHAR I INTEGER (or INT)| SMALLINT | DECIMAL (or DEC)| NUMERIC (or NUM) I FLOAT] REAL | DOUBLE PRECISION | DATE.

column-width

which is enclosed in parentheses, is an integer that specifies the width of the column. (PROC SQL processes this value only for the CHARACTER and VARCHAR data types.)

column-modifier

is one of the following: INFORMAT= | FORMAT= | LABEL= . More than one column-modifier may be specified.

column-constraint

specifies an integrity constraint.

MESSAGE= and MSGTYPE=

specify an error message that is related to an integrity constraint. (Integrity constraints, the column-constraint, MESSAGE=, and MSGTYPE= are not elaborated here, but are discussed in detail later in this chapter.)

Note The entire set of *column-specifications* must be enclosed in parentheses. Multiple *column-specifications* must be separated by commas. Elements within a *column-specification* must be separated by spaces.

Example

Suppose you want to create the temporary table *Work.Discount*, which contains data about discounts that are offered by an airline. There is no existing table that contains the four columns (and column attributes) that you would like to include: **`Destination, BeginDate, EndDate`**, and **`Discount`**. You use the following PROC SQL step to create the table, based on column definitions that you specify:

```
proc sql;
   create table work.discount
          (Destination char(3),
          BeginDate num Format=date9.,
          EndDate num format=date9.,
          Discount num);
```

The SAS log confirms that the table has been created.

Table 5.3: SAS Log

```
NOTE: Table WORK.DISCOUNT created, with 0 rows and 4 columns.
```

Tip In this example and all other examples in this chapter, you are instructed to save your data to a temporary table (in the library *Work)* that will be deleted at the end of the SAS session. To save the table permanently in a different library, use the appropriate libref instead of the libref *Work* in the CREATE TABLE clause.

In the next few sections, you will learn more about specifying data types and column modifiers in a column specification.

Note You will learn to insert rows of data in a table later in this chapter.

Specifying Data Types

When you create a table by defining columns, you must specify a *data type* for each column, following the column name:

column-name ***data-type*** *<(column-width)> <column-modifier-1<…column-modifier-n>>*

For example, the following PROC SQL step (shown also in the previous section) defines four columns: one character column **(`Destination`)** and three numeric columns **(`BeginDate, EndDate`**, and **`Discount`):**

```
proc sql;
   create table work.discount
          (Destination char(3),
          BeginDate num format=date9.,
          EndDate num format=date9.,
          Discount num);
```

SAS tables use two data types: *numeric* and *character*. However, PROC SQL supports additional data types (many, but not all, of the data types that SQL-based databases support). Therefore, in the CREATE TABLE statement, you can specify any of 10 different data types. When the table is created, PROC SQL converts the supported data types that are not SAS data types to either numeric or character format.

Table 5.4: Character Data Types Supported by PROC SQL

Specified Data Type	SAS Data Type
CHARACTER (or *CHAR)*	*CHARACTER*
VARCHAR	*CHARACTER*

Table 5.5: Numeric Data Types Supported by PROC SQL

Specified Data Type	Description	SAS Data Type
NUMERIC (or *NUM)*	floating-point	*NUMERIC*
DECIMAL (or *DEC)*	floating-point	*NUMERIC*
FLOAT	floating-point	*NUMERIC*
REAL	floating-point	*NUMERIC*
DOUBLE PRECISION	floating-point	*NUMERIC*
INTEGER (or *INT)*	integer	*NUMERIC*
SMALLINT	integer	*NUMERIC*
DATE	date	*NUMERIC* with a *DATE.7* informat and format

The following PROC SQL step is very similar to the previous example. The only difference is that this step specifies three supported data types other than CHAR and NUM: VARCHAR, DATE, and FLOAT.

```
proc sql;
   create table work.discount2
          (Destination varchar(3),
          BeginDate date,
          EndDate date,
          Discount float);
```

PROC SQL will convert these data types to either character or numeric, as indicated in the charts above. Therefore, the table *Work.Discount2*(created by this PROC SQL step) and *Work.Discount*(created by the previous PROC SQL step) will contain identical columns.

By supporting data types other than SAS data types, PROC SQL can save you time. In many cases, you can copy native code from an implementation of SQL that is external to SAS without having to modify the data types.

Specifying Column Widths

In SAS, the default *column width* for both character and numeric columns is 8 bytes. However, character and numeric data values are stored differently:

- *Character data* is stored one character per byte.
- *Numeric data* is stored as floating point numbers in real binary representation, which allows for 16- or 17-digit precision within 8 bytes.

PROC SQL allows you to specify a column width for character columns but not for numeric columns.

> **Note** PROC SQL allows the WIDTH and NDEC (decimal places) arguments to be included in the column specification for the DECIMAL, NUMERIC, and FLOAT data types. However, PROC SQL ignores this specification and uses the SAS defaults.

In a column specification, the column width follows the data type and is specified as an integer enclosed in parentheses:

column-name data-type <(column-width)> <column-modifier-1<column-modifier-n>>

In the following PROC SQL step, the first column specification indicates a column width of 3 for the character column **Destination:**

```
proc sql;
   create table work.discount
          (Destination char(3),
          BeginDate num format=date9.,
          EndDate num format=date9.,
          Discount num);
```

Because the last three columns are numeric, no width is specified and these columns will have the default column width of 8 bytes.

Specifying Column Modifiers

In the CREATE TABLE statement, a column specification might include one or more of the following SAS *column modifiers: INFORMAT=, FORMAT=*, and *LABEL=*. Column modifiers, if used, are specified at the end of the column specification.

column-name data-type <(column-width)> <…column-modifier-1<…column-modifier-n>>

Note A fourth SAS column modifier, LENGTH=, is not allowed in a CREATE TABLE clause. It can be used in a SELECT clause.

Example

The following PROC SQL step creates the table *Work.Departments* by specifying 4 columns. The column modifiers LABEL= and FORMAT= are used to specify additional column attributes.

```
proc sql;
   create table work.departments
          (Dept varchar(20) label='Department',
          Code integer label='Dept Code',
          Manager varchar(20),
          AuditDate num format=date9.);
```

The SAS log verifies that the table was created.

Table 5.6: SAS Log

```
NOTE: Table WORK.DEPARTMENTS created, with 0 rows and 4 columns.
```

Displaying the Structure of a Table

Overview

Sometimes you want to look at the structure (the columns and column attributes) of a table that you created or of a table that was created by someone else. When you create a table, the CREATE TABLE statement writes a message to the SAS log, which indicates the number of rows and columns in the table that was created. However, that message does not contain information about column attributes.

If you are working with an existing table that contains rows of data, you can use a PROC SQL query to generate a report that shows all of the columns in a table. However, the report does not list the column attributes, and a PROC SQL query will not generate output for an empty table.

To display a list of columns and column attributes for one or more tables in the SAS log, regardless of whether the tables contain rows of data, you can use the DESCRIBE TABLE statement in PROC SQL.

General form, DESCRIBE TABLE statement:

DESCRIBE TABLE *table-name-1<, … table-name-n>;*

where

table-name

specifies the table to be described as one of the following:

- a one-level name
- a two-level *libref.table* name
- a physical pathname that is enclosed in single quotation marks.

The DESCRIBE TABLE statement writes a CREATE TABLE statement that includes column definitions to the SAS log for the specified table, regardless of how the table was originally created. For example, if the DESCRIBE TABLE statement specifies a table that was created with the DATA step, a CREATE TABLE statement will still be displayed.

Note The DESCRIBE TABLE statement also displays information about any indexes that are defined on a table. You can learn more about using the DESCRIBE TABLE statement to display information about indexes in "Creating and Managing Indexes Using PROC SQL" on page 238.

Tip As an alternative to the DESCRIBE TABLE statement, you can use other SAS procedures, like PROC CONTENTS, to list a table's columns and column attributes. PROC CONTENTS generates a report instead of writing a message to the SAS log, as the DESCRIBE TABLE statement does. You can learn more about using PROC CONTENTS in "Creating Samples and Indexes" on page 470.

Example

Earlier in this chapter, the empty table *Work.Discount* was created by using the CREATE TABLE statement and column specifications shown below:

```
proc sql;
   create table work.discount
          (Destination char(3),
          BeginDate num format=date9.,
          EndDate num format=date9.,
          Discount num);
```

The following DESCRIBE TABLE statement writes a CREATE TABLE statement to the SAS log for the table *Work.Discount*.

```
proc sql;
   describe table work.discount;
```

Note For more information about the BUFSIZE= option, see "Using the BUFSIZE= Option" on page 715.

Table 5.7: SAS Log

```
NOTE: SQL table WORK.DISCOUNT was created like:

create table WORK.DISCOUNT(bufsize=4096)
            (
            Destination char(3),
            BeginDate num format=DATE9.,
            EndDate num format=DATE9.,
            Discount num
            )
```

Creating an Empty Table That Is like Another Table

Overview

Sometimes you want to create a new table that has the same columns and attributes as an existing table, but has no rows. To create an empty table that is like another table, use a *CREATE TABLE statement* with a *LIKE clause*.

General form, CREATE TABLE statement with a LIKE clause:

```
CREATE TABLE table-name
    LIKE table-1;
```

where

table-name

specifies the name of the table to be created.

table-1

specifies the table whose columns and attributes will be copied to the new table.

Example

Suppose you want to create a new table, *Work.Flightdelays2*, that contains data about flight delays. You would like the new table to contain the same columns and attributes as the existing table *Sasuser.Flightdelays*, but you do *not* want to include any of the existing data. The following PROC SQL step uses a CREATE TABLE statement and a LIKE clause to create *Work.Flightdelays2:*

```
proc sql;
   create table work.flightdelays2
      like sasuser.flightdelays;
```

The following DESCRIBE TABLE statement displays the structure of the empty table *Work.Flightdelays2:*

```
proc sql;
   describe table work.flightdelays2;
```

Table 5.8: SAS Log

```
NOTE: SQL table WORK.FLIGHTDELAYS2 was created like:

create table WORK.FLIGHTDELAYS2(bufsize=8192)
  (
   Date num format=DATE9. informat=DATE9.,
   FlightNumber char(3),
   Origin char(3),
   Destination char(3),
   DelayCategory char(15),
   DestinationType char(15),
   DayOfWeek num,
   Delay num
 );
```

Work.Flightdelays2 contains eight columns, as listed.

Specifying a Subset of Columns from a Table

If you want to create an empty table that contains only a specified subset of columns from an existing table, use the SAS data set options *DROP*= or *KEEP*=.

General form, DROP= and KEEP= data set options:

(DROP | KEEP *=column-1< …column-n>)*

where

column

specifies the name of a column to be dropped or kept. Multiple column names must be separated by spaces. The entire option must be enclosed in parentheses.

In the CREATE TABLE statement, the DROP= or KEEP= option can be inserted in either of the following locations:

- between the name of the table that is being created and the LIKE clause (as shown in the following example)
- after the name of the source table, at the end of the LIKE clause.

Example

The following PROC SQL step creates the new table *Work.Flightdelays3* that contains a subset of columns from the table *Sasuser.Flightdelays*. The DROP= option is used to specify that all columns except **`DelayCategory`** and **`DestinationType`** will be included in the new table.

```
proc sql;
   create table work.flightdelays3
          (drop=delaycategory destinationtype)
      like sasuser.flightdelays;
```

For comparison, the results of running the DESCRIBE TABLE statement for the original table and the new table are shown below.

Table 5.9: SAS Log

```
NOTE: SQL table WORK.FLIGHTDELAYS was created like:

create table SASUSER.FLIGHTDELAYS(bufsize=8192)
  (
   Date num format=DATE9. informat=DATE9.,
   FlightNumber char(3),
   Origin char(3),
   Destination char(3),
   DelayCategory char(15),
   DestinationType char(15),
   DayOfWeek num,
   Delay num
 );
```

Table 5.10: SAS Log

```
NOTE: SQL table WORK.FLIGHTDELAYS was created like:

create table WORK.FLIGHTDELAYS3(bufsize=4096)
  (
   Date num format=DATE9. informat=DATE9.,
   FlightNumber char(3),
   Origin char(3),
   Destination char(3),
   DayOfWeek num,
   Delay num
 );
```

As these messages show, *Sasuser.Flightdelays* contains the columns **`DelayCategory`** and **`DestinationType`**, while *Work.Flightdelays3* does not.

Note In PROC SQL, you can apply most of the SAS data set options, such as DROP= and KEEP=, to tables any time that you specify a table. You can use a more limited set of SAS data set options with PROC SQL views. However, because the DROP= and KEEP= options are SAS options and not part of the ANSI standard for SQL, you can use the DROP= and KEEP= options only with the SAS implementation of SQL.

Creating a Table from a Query Result

Overview

Sometimes you want to create a new table that contains both columns and rows that are derived from an existing table or set of tables. In this situation, you can submit one PROC SQL step that does both of the following:

- creates a new table
- populates the table with data from the result of a PROC SQL query.

To create a table from a query result, use a *CREATE TABLE statement* that includes the *keyword AS* and the *clauses that are used in a query*'. SELECT, FROM, and any optional clauses, such as ORDER BY.

General form, CREATE TABLE statement with query clauses:

```
CREATE TABLE table-name AS
       SELECT column-1<, … column-n>
            FROM table-1 | view-1<, … table-n | view-n>
            <optional query clauses>;
```

where

table-name
specifies the name of the table to be created.

SELECT
specifies the column(s) that will appear in the table.

FROM
specifies the table(s) or view(s) to be queried.

optional query clauses
are used to refine the query further and include WHERE, GROUP BY, HAVING, and ORDER BY.

You should be familiar with the use of the CREATE TABLE statement to create a table from a query result. Here is a review of this method.

When a table is created from a query result,

- the new table is populated with data that is derived from one or more tables or views that are referenced in the query's FROM clause
- the new table contains the columns that are specified in the query's SELECT clause
- the new table's columns have the same column attributes (type, length, informat, and format) as those of the selected source columns.

Note When you are creating a table, if you do not specify a column alias for a calculated column, SAS will assign a column name, such as **_TEMA001**.

When query clauses are used within a CREATE TABLE statement, that query's automatic report generation is shut off. Only the new table is generated as output.

Example

Suppose you want to create a new, temporary table that contains data for ticket agents who are employed by an airline. The data that you need is a subset of the data contained in two existing tables, *Sasuser.payrollmaster* and *Sasuser.Staffmaster*. The following PROC SQL step creates the new table *Work.Ticketagents* from the result of a query on the two existing tables. The WHERE clause joins the table by matching EMPID and selects the subset of rows for employees whose **JobCode** contains TA.

```
proc sql;
   create table work.ticketagents as
      select lastname, firstname,
             jobcode, salary
         from sasuser.payrollmaster,
              sasuser.staffmaster
         where payrollmaster.empid
              = staffmaster.empid
              and jobcode contains 'TA';
```

Note Because this query lists two tables in the FROM clause and subsets rows based on a WHERE clause, the query is actually a PROC SQL inner join.

The new table *Work.Ticketagents* is not empty; it contains rows of data. Therefore, you can submit a SELECT statement to display *Work.Ticketagents* as a report:

```
select *
   from work.ticketagents;
```

The first four rows of *Work.Ticketagents* are shown below.

LastName	FirstName	JobCode	Salary

ADAMS	GERALD	TA2	$48.126
AVERY	JERRY	TA3	$54,351
BLALOCK	RALPH	TA2	$45,661
BOSTIC	MARIE	TA3	$54,299

The SAS log also displays a message, indicating that the table has been created.

Table 5.11: SAS Log

```
NOTE: Table WORK.TICKETAGENTS created, with 41 rows and 4 columns.
```

Copying a Table

To copy a table quickly, you can use the CREATE TABLE statement with a query that returns an entire table instead of a subset of columns and rows. The CREATE TABLE statement should contain *only* the following clauses:

- a SELECT clause that specifies that *all* columns from the source table should be selected
- a FROM clause that specifies the source table.

Note Remember that the order of rows in a PROC SQL query result cannot be guaranteed, unless you use an ORDER BY clause. Therefore, a CREATE TABLE statement without an ORDER BY clause can create a table that contains the same rows as the original table, but the rows might be in a different order.

Example

The following PROC SQL step creates the new table *Work.Supervisors2*, which is an exact duplicate of the source table *Sasuser.Supervisors:*

```
proc sql;
   create table work.supervisors2 as
      select *
         from sasuser.supervisors;
```

The first four rows of the two tables are shown below.

Supervisor Id	Stale	Job Category
1677	CT	BC
1834	NY	BC
1431	CT	FA
1433	NJ	FA

Figure 5.1: Source Table: Sasuser.Supervisors

Supervisor Id	State	Job Category
1677	CT	BC
1834	NY	BC
1431	CT	FA
1433	NJ	FA

Figure 5.2: New Table: Work.Supervisors2

Inserting Rows of Data into a Table

Overview

After you have created an empty table, you will want to insert rows of data. You might also want to insert additional rows of data into tables that already contain data. You can use the *INSERT statement* in three different ways to insert rows of data

into existing tables, either empty or populated.

Note You can also use the INSERT statement to insert rows of data in a single table that underlies a PROC SQL view. To learn more about PROC SQL views, see "Creating and Managing Views Using PROC SQL" on page 260.

Method of Inserting Row	Example

insert *values by column name* by using the *SET clause*	`proc sql;`
	`  insert into work.discount`
	`  set destination='LHR',`
	`  begindate='01MAR2000'd,`
	`  enddate='05MAR2000'd,`
	`  discount=.33`
	`  set destination='CPH',`
	`  begindate='03MAR2000'd,`
	`  enddate='10MAR2000'd,`
	`  discount=.15;`

insert *lists of values* by using the *VALUES clause*	`proc sql;`
	`  insert into work.discount (destination,`
	`  begindate,enddate,discount)`
	`  values ('LHR','01MAR2000'd,`
	`  '05MAR2000'd,.33)`
	`  values ('CPH','03MAR2000'd,`
	`  '10MAR2000'd,.15);`

insert rows that are copied from another table *by using a query result*	`proc sql;`
	`  insert into payrollchanges2`
	`  select empid,salary,dateofhire`
	`  from sasuser.payrollmaster`
	`  where empid in ('1919','1350','1401');`

In each method, the INSERT statement inserts new rows of data into the table. To indicate that the rows have been inserted, the SAS log displays a message similar to the following.

Table 5.12: SAS Log

```
NOTE: 1 row was inserted into WORK.DISCOUNT.
```

Here is information on how to use each of these methods to insert rows of data into a table.

Inserting Rows by Using the SET Clause

Sometimes you need to add rows of data to a table, but the data is not currently contained in any table. In this situation, you can use either the SET clause or the VALUES clause in the INSERT statement to specify the data to be added.

The *SET clause* in the *INSERT statement* enables you to specify new data to be added to a table. The SET clause specifies column names and values in pairs. PROC SQL reads each column name-value pair and assigns the value to the specified column. A separate SET clause is used for each row to be added to the table.

The syntax of the INSERT statement that contains the SET clause is shown below.

General form, INSERT statement containing the SET clause:

INSERT INTO *table-name<(target-column-1<, … target-column-n)>*
  SET *column-1=value-1<, … column-n=value-n>*
  <… SETcolumn-1=value-1<, … column-n=value-n>>;

table-name

specifies the name of the table to which rows will be inserted.

target-column

specifies the name of a column into which data will be inserted,

each *SET* clause

specifies one or more values to be inserted in one or more specified columns in a row. Multiple SET clauses are not separated by commas.

column

specifies the name of a column into which data will be inserted.

value

specifies a data value to be inserted into the specified column. Character values must be enclosed in quotation marks.

multiple *column=value* pairs in a SET clause

are separated by commas.

where

Note It is optional to include a list of target column names after the table name in the INSERT TABLE statement that includes a SET clause. The list can include the names of all or only a subset of columns in the table. If you specify an optional list of target column names, then you can specify values *only* for columns that are in the list. You can list target columns in any order, regardless of their position in the table. Any columns that are in the table but *not* listed are given missing values in the inserted rows.

Note Although it is recommended that the SET clause list column-value pairs in order (as they appear in the table or the optional column list), it is *not* required.

Example

Consider the table *Work.Discount*, which was presented in the last topic. *Work.Discount* stores airline discounts for certain flight destinations and time periods in March. By submitting a DESCRIBE TABLE statement, you can see this table's columns and column attributes.

Table 5.13: SAS Log

```
NOTE: SQL table WORK.DISCOUNT was created like:

create table WORK.DISCOUNT(bufsize=4096)
  (
   Destination char(3),
   BeginDate num format=DATE9.,
   EndDate num format=DATE9.,
   Discount num
 );
```

The following PROC SQL step does both of the following:

- adds two rows of new data to *Work.Discount* by using an INSERT statement that contains two SET clauses, one for each row
- generates a report that displays *Work.Discount*, with its two new rows, by using a SELECT statement.

In this situation, you do not need to include an optional list of column names.

```
proc sql;
   insert into work.discount
      set destination='LHR',
          begindate='01MAR2000'd,
          enddate='05MAR2000'd,
          discount=.33
```

```
      set destination='CPH',
          begindate='03MAR2000'd,
          enddate='10MAR2000'd,
          discount=.15;
  select *
    from discount;
```

Destination	BeginDate	EndDate	Discount
LHR	01MAR2000	05MAR2000	0.33
CPH	03MAR2000	10MAR2000	0.15

Inserting Rows by Using the VALUES Clause

The INSERT statement uses the *VALUES clause* to insert a *list of values into a table*. Unlike the SET clause, the VALUES clause does *not* specify a column name for each value, so the values must be listed in the correct order. Values must be specified in the order in which the columns appear in the table or, if an optional column list is specified, in the order in which the columns appear in that list. A separate VALUES clause is used for each row to be added to the table.

General form, INSERT statement containing the VALUES clause:

```
INSERT INTO table-name<(target-column-1<, … target-column-n)>
     VALUES (value-1<, … value-n)>
     <… VALUES(value-1<, … value-n>)>;
```

where

table-name

specifies the name of the table to which rows will be inserted.

target-column

specifies the name of a column into which data will be inserted,

each *VALUES* clause

lists the values to be inserted in some or all columns in one row, which is enclosed in parentheses. Multiple VALUES clauses are not separated by commas.

value

specifies a data value to be added. Character values must be enclosed in quotation marks. Multiple values must be separated by commas. Values must be listed in positional order, either as they appear in the table or, if the optional column list is specified, as they appear in the column list.

Note It is optional to include a list of target column names after the table name in the INSERT TABLE statement that includes a VALUES clause. The list can include the names of all or only a subset of columns in the table. If an optional list of target column names is specified, then only those columns are given values by the statement. Target columns can be listed in any order, regardless of their position in the table. Any columns that are in the table but *not* listed are given missing values in the inserted rows.

You can use the VALUES clause to insert a value for all or only some of the columns in the table.

If you want to …	Then…	Example

insert a value for *all columns* in the table	You can *omit the optional list of column names* in the INSERT statement.
	PROC SQL
	▪ reads values in the order in which they are specified in the VALUES clause
		`insert into work.newtable`
		`values ('WI','FLUTE',6)`
		`values ('ST','VIOLIN',3);`

	▪ inserts the values into columns *in the order in which the columns appear in the table*.	
insert a value for only *some of the columns* in the table	You must *include a list of column names* in the INSERT statement. PROC SQL ▪ reads values in the order in which they are specified in the VALUES clause ▪ inserts the values into columns *in the order in which the columns are specified in the column list*.	`insert into work.newtable (item,qty) values ('FLUTE',6) values ('VIOLIN',3);`

You *must list a valuefor every column* into which PROC SQL will insert values (as specified in either the table or the optional list of column names). To specify that a value is missing, use a space enclosed in single quotation marks for character values and a period for numeric values. For example, the following VALUES clause specifies values to be inserted in three columns; the first two values are missing:

```
values (' ', ., 45)
```

In this example, the first value specified is a missing value for a character column, and the second value is a missing value for a numeric column.

Example

Suppose you want to insert two more rows into the table *Work.Discount*, which stores airline discounts for certain flight destinations and time periods in March. In the previous section, you inserted two rows into *Work.Discount* by using the SET clause, so the table now looks like the following table.

Destination	BeginDate	EndDate	Discount
LHR	01MAR2000	05MAR2000	0.33
CPH	03MAR2000	10MAR2000	0.15

Add two more rows, by using the VALUES clause. The following PROC SQL step adds two rows of new data to *Work.Discount* and generates a report that displays the updated table:

```
proc sql;
   insert into work.discount (destination,
          begindate,enddate,discount)
      values ('ORD','05MAR2000'd,'15MAR2000'd,.25)
      values ('YYZ','06MAR2000'd,'20MAR2000'd,.10);
   select *
      from work.discount;
```

Destination	BeginDate	EndDate	Discount:
LHR	01MAR2000	05MAR2000	0.33
CPH	03MAR2000	10MAR2000	0.15
ORD	05MAR2000	15MAR2000	0.25
YYZ	06MAR2000	20MAR2000	0.1

The two rows that were just inserted by using the VALUES clause are the third and fourth rows above.

You might have noticed that the INSERT statement in this example includes an optional list of column names. In this example, data is being inserted into *all* columns of the table, and the values are listed in the order in which the columns appear in the table, so it is not strictly necessary to use a column list. However, including the list of column names makes it easier to read the code and understand what the code is doing.

Inserting Rows from a Query Result

The fastest way to insert rows of data into a table is to use a query to select existing rows from one or more tables (or views) and to insert the rows into another table. You can insert rows from a query result into either an empty table or a table that already contains rows of data. When you add rows of data to a table that already contains rows, the new rows

are added at the end of the table.

To insert rows from a query result, use an *INSERT statement* that includes the *clauses that are used in a query:* SELECT, FROM, and any optional clauses, such as ORDER BY. Values from the query result are inserted into columns in the order in which the columns appear in the table or, if an optional column list is specified, in the order in which the columns appear in that list.

General form, INSERT statement containing query clauses:

```
INSERT INTO table-name<(target-column-1<,… target-column-n)>
     SELECT column-1<, … column-n>
          FROM table-1 | view-1<, …table-n | view-n>
          <optional query clauses>;
```

where

table-name

specifies the name of the table to which rows will be inserted.

target-column

specifies the name of a column into which data will be inserted.

SELECT

specifies the column(s) that will be inserted.

FROM

specifies the table(s) or view(s) to be queried.

optional query clauses

are used to refine the query further. These include the WHERE, GROUP BY, HAVING, and ORDER BY clauses.

Note It is optional to include a list of target column names after the table name in the INSERT TABLE statement that includes query clauses. The list can include the names of all or only a subset of columns in the table. If an optional list of target column names is specified, then only those columns are given values by the statement. Target columns might be listed in any order, regardless of their position in the table. Any columns that are in the table but *not* listed are given missing values in the inserted rows.

Example

A mechanic at a company has been promoted from level 2 to level 3, and you need to add this employee to *Sasuser.Mechanicslevel3*, a table that lists all level-3 mechanics. Create a temporary copy of *Sasuser.Mechanicsleve3* called *Work.Mechanicslevel3_New*, and display the new table in a report:

```
proc sql;
   create table work.mechanicslevel3_new
      select *
         from sasuser.mechanicslevel3;
```

EmpID	JobCode	Salary
1499	ME3	$60,235
1409	ME3	$58,171
1379	ME3	$59,170
1521	ME3	$58,136
1385	ME3	$61,460
1420	ME3	$60,299

1882	ME3	$58,153

Next, you insert a row into *Work.Mechanicslevel3_New* for the new level-3 employee, whose **EmpID** is *1653*. This employee is currently listed in *Sasuser.Mechanicslevel2*, so your INSERT statement queries the table *Sasuser.Mechanicslevel2*. Your PROC SQL step ends with a SELECT statement that outputs the revised table Work.Mechanicslevel3_New to a report.

```
proc sql;
   insert into work.mechanicslevel3_new
      select empid, jobcode, salary
         from sasuser.mechanicslevel2
         where empid='1653';
      select *
         from work.mechanicslevel3_new;
```

EmpID	JobCode	Salary
1499	ME3	$60,235
1409	ME3	$58,171
1379	ME3	$59,170
1521	ME3	$58,136
1385	ME3	$61,460
1420	ME3	$60,299
1882	ME3	$58,153
1653	ME2	$49,151

The row that you have inserted into *Work.Mechanicslevel3_New* is row 8 above. As you can see, the values for **JobCode** and **Salary** for the new employee will have to be changed. Updating existing values in a table is covered later in this chapter.

Note Although the new row is shown here at the bottom of the table, the order of rows in a PROC SQL query cannot be guaranteed if an ORDER BY clause is not used.

Creating a Table That Has Integrity Constraints

Overview

Integrity constraints are rules that you can specify in order to restrict the data values that can be stored for a column in a table. SAS enforces integrity constraints when values associated with a column are added, updated, or deleted. Integrity constraints help you preserve the validity and consistency of your data.

You can create integrity constraints by using either *PROC SQL* or *PROC DATASETS*. PROC DATASETS can assign constraints only to an existing table. PROC SQL can assign constraints either as it creates a new table or as it modifies an existing table. This chapter discusses the use of PROC SQL to create integrity constraints while creating a table.

Note To learn more about the use of PROC DATASETS to create integrity constraints, see "Modifying SAS Data Sets and Tracking Changes" on page 656. For additional information about integrity constraints, see the SAS documentation.

Note To add integrity constraints to an existing table using PROC SQL, use the ALTER TABLE statement.

When you place integrity constraints on a table, you specify the type of constraint that you want to create. Each constraint has a different action.

Constraint Type	Action
CHECK	Ensures that a specific set or range of values are the only values in a column. It can also check the validity of a value in

	one column based on a value in another column within the same row.
NOTNULL	Guarantees that a column has non-missing values in each row.
UNIQUE	Enforces uniqueness for the values of a column.
PRIMARY KEY	Uniquely defines a row within a table, which can be a single column or a set of columns. A table can have only one PRIMARY KEY. The PRIMARY KEY includes the attributes of the constraints NOT NULL and UNIQUE.
FOREIGN KEY	Links one or more rows in a table to a specific row in another table by matching a column or set of columns (a FOREIGN KEY) in one table with the PRIMARY KEY in another table. This parent/child relationship limits modifications made to both PRIMARY KEY and FOREIGN KEY constraints. The only acceptable values for a FOREIGN KEY are values of the PRIMARY KEY or missing values.

Note When you add an integrity constraint to a table that contains data, SAS checks all data values to determine whether they satisfy the constraint before the constraint is added.

You can use integrity constraints in two ways, *general* and *referential*. General constraints enable you to restrict the data values accepted for a column in a single table.

Referential constraints enable you to link the data values of a column in one table to the data values of columns in another table.

General Integrity Constraints

General integrity constraints enable you to restrict the values of columns within a single table. The following four integrity constraints can be used as general integrity constraints:

- CHECK
- NOT NULL
- UNIQUE
- PRIMARY KEY.

Note A PRIMARY KEY constraint is a general integrity constraint if it does not have any FOREIGN KEY constraints referencing it. A PRIMARY KEY used as a general constraint is a shortcut for assigning the constraints NOT NULL and UNIQUE.

Referential Integrity Constraints

A referential integrity constraint is created when a PRIMARY KEY integrity constraint in one table is referenced by a FOREIGN KEY integrity constraint in another table. There are two steps that must be followed to create a referential integrity constraint:

1. Define a PRIMARY KEY constraint on the first table.
2. Define a FOREIGN KEY constraint on other tables.

Note Integrity constraints

- follow ANSI standards
- cannot be defined for views
- cannot be defined for historical versions of generation data sets.

To create a table that has integrity constraints, use a CREATE TABLE statement that specifies both columns and constraints. There are two ways to specify integrity constraints in the CREATE TABLE statement:

- in a column specification
- as a separate constraint specification.

You can use either or both of these methods in the same CREATE TABLE statement.

Creating a Constraint in a Column Specification

Earlier in this chapter, you learned how to create a table by using a CREATE TABLE statement that contains column specifications:

```
CREATE TABLE table-name
       (column-specification-1<,
       …column-specifieation-n>);
```

You also learned that a column specification consists of these elements:

column-definition <*column-constraint-1*<, …*column-constraint-n*>>

<MESSAGE='*message-string*'<MSGTYPE=*message-type*>>

The column specifications used in earlier examples contained only the column definition. Now we will learn how to create an integrity constraint with a column, by specifying the optional *column constraint* in the column specification:

General form, column-constraint in a column-specification:

column-definition <*column-constraint-1*<, …*column-constraint-n*>>

<MESSAGE='*message-string*'<MSGTYPE=*message-type*>>

where

column-constraint

is one of the following:

CHECK *(expression)*

specifies that all rows in the table (which is specified in the CREATE TABLE statement) satisfy the *expression*, which can be any valid where-expression.

DISTINCT

specifies that the values of the column must be unique within the table. This constraint is identical to UNIQUE.

NOT NULL

specifies that the column does not contain a null or missing value, including special missing values.

PRIMARY KEY

specifies that the column is a PRIMARY KEY column, that is, a column that does not contain missing values and whose values are unique.

REFERENCES *table-name* <ON DELETE *referential-action*> <ON UPDATE *referential-action*>

specifies that the column is a FOREIGN KEY, that is, a column whose values are linked to the values of the PRIMARY KEY column in another table (the *table-name* that is specified for *REFERENCES)*. The *referential-actions* are performed when the values of a PRIMARY KEY column that is referenced by the FOREIGN KEY are updated or deleted. The *referential-action* specifies the type of action to be performed on all matching FOREIGN KEY values and is one of the following:

CASCADE

allows PRIMARY KEY data values to be updated, and updates matching values in the FOREIGN KEY to the same values.

Note This referential action is currently supported for updates only.

RESTRICT

occurs only if there are matching FOREIGN KEY values. This referential action is the default.

SET NULL

sets all matching FOREIGN KEY values to NULL.

UNIQUE
specifies that the values of the column must be unique within the table. This constraint is identical to DISTINCT.

Note The optional MSGTYPE= and MESSAGE= elements will be discussed later in this chapter.

Just like a column, an integrity constraint must have a unique name within the table. When you create an integrity constraint by specifying a column constraint in a column specification, then SAS automatically assigns a name to the constraint. The form of the constraint name depends on the type of constraint, as shown below:

Constraint Type	Default Name
CHECK	**_CKxxxx_**
FOREIGN KEY	**_FKxxxx_**
NOT NULL	**_NMxxxx_**
PRIMARY KEY	**_PKxxxx_**
UNIQUE	**_UNxxxx_**

Note **xxxx** is a counter that begins at 000l.

Here is an example of a PROC SQL step that creates integrity constraints by specifying one or more column constraints in a column specification.

Example

Suppose you need to create the table *Work.Employees* to store the identification number, name, gender, and hire date for all employees. In addition, you want to ensure that

- the **ID** column contains only values that are nonmissing and unique
- the **Gender** column contains only the values *M* and *F*.

The following PROC SQL step creates the table *Work.Employees*, which contains four columns and integrity constraints for two of those columns:

```
proc sql;
   create table work.employees
      (ID char (5) primary key,
      Name char(10),
      Gender char(1) not null check(gender in ('M','F')),
      HDate date label='Hire Date');
```

In the column specification for **ID**, the PRIMARY KEY column constraint ensures that the **ID** column will contain only values that are nonmissing and unique.

The column specification for **Gender** defines two integrity constraints:

- The NOT NULL column constraint ensures that the values of **Gender** will be nonmissing values.
- The CHECK column constraint ensures that the values of **Gender** will satisfy the expression **gender in ('M',F')**.

Here is another method of creating integrity constraints: specifying a constraint specification in the CREATE TABLE statement.

Creating a Constraint by Using a Constraint Specification

Sometimes you might prefer to create integrity constraints outside of column specifications, by specifying individual constraint specifications in the CREATE TABLE statement:

```
CREATE TABLE table-name
    (column-specification-1<,
```

```
…colmnn-specification-n><,
constraint-specification-1 ><,
…constraint-specification-n>);
```

The first specification in the CREATE TABLE statement must be a column specification. However, following the initial column specification in the statement, you can include multiple additional column specifications, constraint specifications, or both. All specifications after the first column specification can be listed in any order. The entire list of column specifications and constraint specifications follows the same guidelines that were presented earlier for column specifications:

- The entire set of column specifications and constraint specifications must be enclosed in parentheses.
- Multiple column specifications and constraint specifications must be separated by commas.

There are several important *differences* between specifying an integrity constraint within a column specification and specifying an integrity constraint by using a separate constraint specification. Using a constraint specification offers the following advantages:

- You can specify a name for the constraint. In fact, you must specify a name, because SAS does not automatically assign one.
- For certain constraint types, you can define a constraint for multiple columns in a single specification.

The syntax of a *constraint specification* is shown below.

CONSTRAINT *constraint-name constraint* <MESSAGE ='*message-string*' <MSGTYPE =*message-type*>>

constraint-name

specifies a name for the constraint that is being specified. The name must be a valid SAS name.

Caution PRIMARY, FOREIGN, MESSAGE, UNIQUE, DISTINCT, CHECK, and NOT *cannot* be used as values for *constraint-name*.

constraint

is one of the following:

CHECK *(expression)*

specifies that all rows in *table-name*(which is specified in the CREATE TABLE statement) satisfy the *expression*, which can be any valid where-expression.

DISTINCT *(column-1<, …column-n>)*

specifies that the values of each *column* must be unique within the table. This constraint is identical to UNIQUE.

FOREIGN KEY *(column-1<,…column-n>)*

REFERENCES *table-name*

<ON DELETE *referential-action*>

<ON UPDATE *referential-action*>

specifies a FOREIGN KEY, that is, a set of *columns* whose values are linked to the values of the PRIMARY KEY column in another table (the *table name* that is specified for *REFERENCES)*. The *referential-actions* are performed when the values of a PRIMARY KEY column that is referenced by the FOREIGN KEY are updated or deleted. The *referential-action* specifies the type of action to be performed on all matching FOREIGN KEY values, and is one of the following:

- **CASCADE**

  allows PRIMARY KEY data values to be updated, and updates matching values in the FOREIGN KEY to the same values.

Note This referential action is currently supported for updates only.

- **`RESTRICT`**

  occurs only if there are matching FOREIGN KEY values. This referential action is the default.

- **`SET NULL`**

  sets all matching FOREIGN KEY values to NULL.

NOT NULL *(column)*

specifies that the *column* does not contain a null or missing value, including special missing values.

PRIMARY KEY *(column-1<, … column-n>)*

specifies one or more *columns* as PRIMARY KEY columns, that is, columns that do not contain missing values and whose values are unique.

UNIQUE *(column-1<, … column-n>)*

specifies that the values of each *column* must be unique within the table. This constraint is identical to DISTINCT.

MESSAGE=

specifies a *message-string* that specifies the text of an error message that is written to the SAS log when the integrity constraint is not met. The maximum length of message-string is 250 characters.

MSGTYPE=

specifies the *message-type*, which specifies how the error message is displayed in the SAS log when an integrity constraint is not met. The message-type is one of the following:

NEWLINE	the text that is specified for *MESSAGE*= is displayed in addition to the default error message for that integrity constraint.
USER	only the text that is specified for *MESSAGE*= is displayed.

Note Elements within a constraint-specification must be separated by spaces.

You might have noticed another difference between the two methods of creating an integrity constraint. When you use a column specification to create a FOREIGN KEY integrity constraint, you use the keyword FOREIGN KEY in addition to the keyword REFERENCES.

Here is an example of a PROC SQL step that uses column specifications to create integrity constraints on a column.

Example

In an example earlier in this chapter, the table *Work.Discount* was created to hold data about discounts that are offered by an airline. Suppose you now want to ensure that the table

- holds only discounts that are less than or equal to .5
- does not allow missing values for **`Destination`**.

Create a new version of the table *Work.Discount*, called *Work.Discount3*, that includes two integrity constraints. One integrity constraint limits the values that can be entered in the **`Discount`** column and the other prevents missing values from being entered in the **`Destination`** column. The following PROC SQL step creates *Work.Discount3* by specifying four columns and two integrity constraints:

```
proc sql;
   create table work.discount3
          (Destination char(3),
          BeginDate num Format=date9.,
          EndDate num format=date9.,
          Discount num,
          constraint ok_discount check (discount le .5),
          constraint notnull_dest not null(destination));
```

The CHECK integrity constraint named **OK_Discount** uses the WHERE expression **discount le .5** to limit the values that can be added to the **Discount** column.

The NOT NULL integrity constraint named **NotNull_Dest** prevents missing values from being entered in **Destination**.

Handling Errors in Row Insertions

Overview

When you add rows to a table that has integrity constraints, PROC SQL evaluates the new data to ensure that it meets the conditions that are determined by the integrity constraints. If the new (or modified) data complies with the integrity constraints, the rows are added. However, if you add data that does *not* comply with the integrity constraints, the rows are *not* added. To find out whether rows of data have been successfully added, you need to check the SAS log.

Note PROC SQL also evaluates changes that are made to existing data by using the UPDATE and DELETE statements. These statements are discussed later in this chapter.

Example

In a previous section of this chapter, the following PROC SQL step was used to create the table *Work.Discount3* with two integrity constraints, one on the column **Discount** and the other on the column **Destination:**

```
proc sql;
   create table work.discount3
          (Destination char(3),
          BeginDate num Format=date9.,
          EndDate num format=date9.,
          Discount num,
          constraint ok_discount check (discount le .5),
          constraint notnull_dest not null(destination));
```

This table does not yet contain any rows, so add some data. The following INSERT statement uses the VALUES clause to add two rows of data to the table:

```
proc sql;
   insert into work.discount3
          values('CDG','03MAR2000'd,'10MAR2000'd,.15)
          values('LHR','10MAR2000'd,'12MAR2000'd,.55);
```

When this PROC SQL step is submitted, the following messages are displayed in the SAS log.

Table 5.14: SAS Log

```
ERROR: Add/Update failed for data set WORK.DISCOUNT3
because data value(s) do not comply with integrity constraint
ok_discount.
NOTE: This insert failed while attempting to add data from
VALUES clause 2 to the data set.
NOTE: Deleting the successful inserts before error noted above
to restore table to a consistent state.
```

The three parts of this message explain what the problem is:

- The error message indicates that this attempt to add *rows failed*. One or more of the data values for **Discount** does not comply with the integrity constraint **OK_Discount**, which specifies that values in the column **Discount** must be less than or equal to *.5*.
- The first note indicates that there is a problem with the second VALUES clause. This clause specifies a value of .55 for the column **Discount**, which does *not* comply.

  Caution Even if multiple VALUES clauses specify non-compliant data values, the SAS log lists only the *first* VALUES clause that violates the constraint.

- The second note indicates that SAS is "deleting the successful inserts" before the error. Even though all the other

specified data is valid, *none* of the data has been added to the table.

We need to consider why SAS prevented any of the data from being added to the table.

Using the UNDO_POLICY= Option to Control UNDO Processing

When you use the INSERT or UPDATE statement to add or modify data in a table, you can control how PROC SQL handles updated data if any errors occur during the insertion or update process. You can use the *UNDO_POLICY*= option in the PROC SQL statement to specify whether PROC SQL will make or undo the changes you submitted up to the point of the error.

You can specify one of the following values for the UNDO_POLICY= option.

UNDO_POLICY=Setting	Description
REQUIRED	PROC SQL performs UNDO processing for INSERT and UPDATE statements. If the UNDO operation cannot be done reliably, PROC SQL *does not execute the statement* and issues an ERROR message. This is the PROC SQL default.
NONE	PROC SQL *skips records that cannot be inserted or updated*, and writes a warning message to the SAS log similar to that written by PROC APPEND. Any data that meets the integrity constraints *is* added or updated.
OPTIONAL	PROC SQL *performs UNDO processing if it can be done reliably*. If the UNDO cannot be done reliably, then no UNDO processing is attempted. This action is a combination of REQUIRED and NONE. If UNDO can be done reliably, then it is done, and PROC SQL proceeds as if UNDO_POLICY=REQUIRED is in effect. Otherwise, it proceeds as if UNDO_POLICY=NONE was specified.

Caution In the following two situations, you cannot reliably attempt the UNDO operation:

A SAS data set that is accessed through a SAS/SHARE server and opened with CNTLLEV=RECORD can allow other users to update newly inserted records. An error during the insert deletes the record that the other user updated.

Changes made through a SAS/ACCESS view might not be able to reverse changes made by the INSERT or UPDATE statement without reversing other changes at the same time.

Note The ANSI standard for SQL includes a ROLLBACK statement that is used for UNDO processing. The ROLLBACK statement is not currently supported in PROC SQL.

Note When you use the UNDO_POLICY= option, the value that you set remains in effect for the entire PROC SQL statement or until a RESET statement is used to change the option. To learn more about the RESET statement, see "Managing Processing Using PROC SQL" on page 278.

Example

In the last example, the INSERT step was used to insert two rows of data into the table *Work.Discount3*, which has two integrity constraints. Because the UNDO_POLICY= option was not specified in the code, PROC SQL used the default policy, which is UNDO_POLICY=REQUIRED. When PROC SQL encountered a value in the INSERT statement that violated an integrity constraint, *none* of the new values specified in the INSERT statement were added to the table.

Consider what happens when we submit the same INSERT statement and specify the option *UNDO_POLICY=NONE*.

The following PROC SQL step creates the table *Work.Discount4*, which has four columns and two integrity constraints, and inserts the same two rows of data that were inserted in the earlier example. In this case, however, the option *UNDO_POLICY=NONE* is specified.

```
proc sql undo_policy=none;
   create table work.discount4
          (Destination char(3),
          BeginDate num Format=date9.,
          EndDate num format=date9.,
          Discount num,
          constraint ok_discount check (discount le .5),
          constraint notnull_dest not null(destination));
   insert into work.discount4
```

```
        values('CDG','03MAR2000'd,'10MAR2000'd,.15)
        values('LHR','10MAR2000'd,'12MAR2000'd,.55);
```

As you know, one of the data values for the column **Discount** violates the specified constraint. When this step is submitted, the SAS log displays the following messages.

Table 5.15: SAS Log

```
WARNING: The SQL option UNDO_POLICY=REQUIRED is not in effect.
If an error is detected when processing this INSERT statement,
that error will not cause the entire statement to fail.
ERROR: Add/Update failed for data set WORK.DISCOUNT4 because
data value(s) do not comply with integrity constraint ok_discount.
NOTE: This insert failed while attempting to add data from VALUES
clause 2 to the data set.

NOTE: 2 rows were inserted into WORK.DISCOUNT4 -- of these 1 row
was rejected as an ERROR, leaving 1 row that was inserted
successfully.
```

The four parts of this message explain what the problem is and how PROC SQL will

handle UNDO processing:

- The warning tells you that you have specified a setting for the UNDO_POLICY= option that is different from the default (REQUIRED). The warning also explains that, as a result, if an error is detected, the error will *not* cause the entire INSERT statement to fail.
- The error message was also displayed in the earlier example, when the default setting of UNDO_POLICY= was in effect. This message states that the INSERT statement failed and explains why.
- The first note was also displayed in the earlier example, when the default setting of UNDO_POLICY= was in effect. This note identifies the first VALUES clause that contains non-compliant data.
- The second note tells you that one row (the first row of the two rows that you specified) was inserted successfully into the table.

Displaying Integrity Constraints for a Table

Overview

Sometimes you want to add data to a table but you are not sure what integrity constraints, if any, the table has. To display only the integrity constraints for a specified table, usea *DESCRIBE TABLE CONSTRAINTS statement*. (The DESCRIBE TABLE statement, which is discussed earlier in this chapter, lists both a CREATE TABLE statement and the table's integrity constraints in the SAS log.)

Note Some versions of SAS display information about integrity constraints in output as well as in the SAS log.

General form, DESCRIBE TABLE CONSTRAINTS statement:

DESCRIBE TABLE CONSTRAINTS *table-name-1<, … table-name-n>;*

where

table-name

specifies the table to be described as one of the following:

- a one-level name
- a two-level *libref.table* name
- a physical pathname that is enclosed in single quotation marks.

Example

To display only the table constraints for the table *Work.Discount4* that was created earlier, you submit the following PROC SQL step:

```
proc sql;
   describe table constraints work.discount4;
```

Table 5.16: SAS Log

```
NOTE: SQL table WORK.DISCOUNT4 ( bufsize=4096 ) has the
following integrity constraint(s):

-----Alphabetic List of Integrity Constraints-----

      Integrity                              Where
   *  Constraint     Type     Variables     Clause
-------------------------------------------------
1     notnull_dest Not Null Destination
2     ok_discount Check Discount<=0.5
```

---Alphabetic List of Integrity Constraints---				
#	Integrity Constraint	Type	Variables	Where Clause
1	notnull_dest	Not Null	Destination	
2	ok_discount	Check		Discounted<=0.5

As shown, *Work.Discount4* has two integrity constraints: **`NotNull_Dest`** and **`OK_Discount`**.

Updating Values in Existing Table Rows

Overview

To modify data values in some or all of the existing rows in a table, you use the *UPDATE statement* in PROC SQL. In the UPDATE statement, for each column whose rows you want to modify, you specify an *expression* that indicates how the values should be modified. For example, the following expression indicates that the values for the column **`Units`** should be multiplied by 4:

```
units=units*4
```

You can use the UPDATE statement in two main ways.

Method of Updating Table	Example

update all (or a subset of) rows in a column with the *same expression*	`proc sql;`
	`  update work.payrollmaster_new`
	`  set salary=salary*1.05`
	`  where jobcode like '__1';`

update different rows in a column with *different expressions*	`proc sql;`
	`  update work.payrollmaster_new`
	`  set salary=salary*`
	`  case when substr(jobcode,3,1)='1'`
	`  then 1.05`
	`  when substr(jobcode,3,1)='2'`
	`  then 1.10`
	`  when substr(jobcode,3,1)='3'`
	`  then 1.15`
	`  else 1.08`
	`  end;`

Note The UPDATE statement does *not* insert new rows into the table. To insert rows, you must use the INSERT

statement.

Note You can also use the UPDATE statement to update existing values in a table that underlies a PROC SQL view. For details, see "Creating and Managing Views Using PROC SQL" on page 260.

We will consider each of these methods for updating existing rows in a table.

Updating Rows by Using the Same Expression

To update all (or a subset of) rows in a column with the *same expression*, use an *UPDATE statement* that contains a *SET clause* and (optionally) a *WHERE clause*.

General form, basic UPDATE statement for updating table rows:

```
UPDATE table-name
    SET column-1=expression<, … column-n=expression>>
    <WHEREexpre&sion>;
```

where

table-name

specifies the name of the table in which values will be updated.

SET

specifies one or more pairs of *column* names to be updated, and *expressions* that indicate how each column is to be updated.

WHERE

is optionally used to specify an *expression* that subsets the rows to be updated.

Caution If you want to update only a subset of rows in the table, you must specify a WHERE clause or *all* rows of the table will be updated.

Example

Suppose a company is considering giving all level-1 employees a 5% raise. Employee salaries are stored in the table *Sasuser.payrollmaster*. You do not want to update the original table, so you create a temporary copy of *Sasuser.payrollmaster*, called *Work.Payrollmaster_New*. The following PROC SQL step creates *Work.Payrollmaster_New* based on a query result and generates an output report of the new table:

```
proc sql;
   create table work.payrollmaster_new as
      select *
         from sasuser.payrollmaster;
   select *
      from work.payrollmaster_new;
```

The first 10 rows of *Work.Payrollmaster_New*, the table in which you will update salaries, are shown below.

DateOfBirth	DateOfHire	EmpID	Gender	JobCode	Salary
16SEP1958	07JUN1985	1919	M	TA2	$48,126
19OCT1962	12AUG1988	1653	F	ME2	$49,151
08NOV1965	19OCT1988	1400	M	ME1	$41,677
04SEP1963	01AUG1988	1350	F	FA3	$46,040
16DEC1948	21NOV1983	1401	M	TA3	$54,351
29APR1952	11JUN1978	1499	M	ME3	$60,235
09JUN1960	04OCT1988	1101	M	SCP	$26,212

03APR1959	14FEB1979	1333	M	PT2	$124,048
20JAN1961	05DEC1988	1402	M	TA2	$45,661
26DEC1966	09OCT1987	1479	F	TA3	$54,299

Next, you write a PROC SQL step that updates the specified rows. The UPDATE statement contains both of the following:

- a SET clause that specifies the expression to be used in updating **`Salary`**
- a WHERE clause that specifies a subset of rows (level-1 employees) to be updated.

```
proc sql;
   update work.payrollmaster_new
      set salary=salary*1.05
      where jobcode like '__1';
```

Finally, you can use a SELECT statement to display the updated table as a report. The first 10 rows of *Work.Payrollmaster_New*, with updates, are shown below.

DateOfBirth	DateOfHire	EmpID	Gender	JobCode	Salary
16SEP1958	07JUN1985	1919	M	TA2	$48,126
19OCT1962	12AUG1988	1653	F	ME2	$49,151
08NOV1965	19OCTI988	1400	M	ME1	$43,761
04SEP1963	01AUG1988	1350	F	FA3	$46,040
16DEC1948	21NOV1983	1401	M	TA3	$54,351
29APR1952	11JUN1978	1499	M	ME3	$60,235
09JUN1960	04OCT1988	1101	M	SCP	$26,212
03APR1959	14FEB1979	1333	M	PT2	$124,048
20JAN1961	05DEC1988	1402	M	TA2	$45,661
26DEC1966	09OCT1987	1479	F	TA3	$54,299

The third row lists data for a level-1 employee, and that person's salary has been updated.

If you wanted to increase *all* of the salaries, you would simply remove the WHERE clause from the UPDATE statement:

```
proc sql;
   update work.payrollmaster_new
      set salary=salary*1.05;
```

Updating Rows by Using Different Expressions

Sometimes you want to use different expressions to modify values for different subsets of rows within a column.

For example, instead of only raising the salary of level-1 employees by 5%, you might also want to raise the salaries of level-2 employees by 10%, and so on, using a different percentage increase for each group of employees.

There are two possible ways to use different expressions to update different subsets of rows.

Method of Updating Table	Example

use *multiple UPDATE statements* subset of rows
A single UPDATE statement can contain only a single WHERE clause, so multiple UPDATE statements are needed to specify expressions for multiple subsets of rows.
	`proc sql;`
	`  update work.payrollmaster_new`
	`  set salary=salary*1.05`
	`  where substr(jobcode,3,1)='1';`
	`  update work.payrollmaster_new`
	`  set salary=salary*1.10`
	`  where substr(jobcode,3,1)='2';`
	`  update work.payrollmaster_new`
	`  set salary=salary*1.15`
	`  where substr(jobcode,3,1)='3';`

use a *single UPDATE statement* that contains a *CASE expression*

```
proc sql;
   update work.payrollmaster_new
      set salary=salary*
         case
            when substr(jobcode,3,1)='1'
               then 1.05
            when substr(jobcode,3,1)='2'
               then 1.10
            when substr(jobcode,3,1)='3'
               then 1.15
            else 1.08
         end;
```

The first method, which requires the use of multiple UPDATE statements, is cumbersome because the SET statement and expression must be repeated in each UPDATE statement. In this example, the first method is inefficient because the table *Work.Payrollmaster_New* must be read three times.

The second method, which uses conditional processing (the CASE expression), is recommended. We will now consider the second method.

To update different subsets of rows in a table in different ways, you can incorporate conditional processing by using the *CASE expression* in the SET clause of an UPDATE statement. The CASE expression selects result values that satisfy specified conditions.

General form, CASE expression:

```
CASE <case-operand>
     WHEN when-condition THEN result-expression
     <...WHENwhen-conditionTHENresult-expression>
     <ELSEresult-expression>
END;
```

where

CASE
: performs conditional processing.

case-operand
: is an optional expression that resolves to a table column whose values are compared to all the *when-conditions*.

WHEN
: specifies a *when-condition*, a shortened expression that assumes *case-operand* as one of its operands, and that resolves to true or false.

THEN
: specifies a *result-expression*, an expression that resolves to a value.

ELSE
: specifies a *result-expression*, which provides an alternate action if none of the *when-conditions* is executed.

END
: indicates the end of the CASE expression.

Caution Although the ELSE clause is optional, its use is strongly recommended. If you omit the ELSE clause, each row that is *not* described in one of the WHEN clauses receives a *missing value* for the column that you are updating.

Note You can also use the CASE expression in the INSERT and SELECT statements.

Example

In the following UPDATE statement, the CASE expression contains three WHEN-THEN clauses that specify three different subsets of rows in the table *Work.Insure_New*.

- homeowners that are insured by Acme
- homeowners that are insured by Reliable
- homeowners that are insured by Homelife.

```
update work.insure_new
   set pctinsured=pctinsured*
      case
         when company='ACME'
             then 1.10
         when company='RELIABLE'
             then 1.15
         when company='HOMELIFE'
             then 1.25
        else 1
    end;
```

PROC SQL updates each specified subset of rows differently, according to the corresponding WHEN-THEN (or ELSE) clause.

How PROC SQL Updates Rows Based on a CASE Expression

When you specify a CASE expression, PROC SQL updates each row as follows:

1. In the CASE expression, PROC SQL finds the WHEN-THEN clause that contains a condition that the row matches.
2. The CASE expression then returns the result from the matching WHEN-THEN clause to the SET clause. The returned value completes the expression in the SET clause.
3. The SET clause uses the completed expression to update the value of the specified column in the current row.

The use of the CASE expression is efficient because of the way PROC SQL processes the WHEN-THEN clauses. The WHEN-THEN clauses in the CASE expression are evaluated sequentially. When a matching case is found, the THEN expression is evaluated and set, and the remaining WHEN cases are *not* considered.

How the Case Operand Works

In the next few sections, you will learn about the use of the CASE expression in the UPDATE statement, without and with the optional case operand:

CASE <*case-operand*>

Updating Rows by Using the CASE Expression without a Case Operand

Here is an example of an UPDATE statement that uses the CASE expression for conditional processing. This example shows the form of the CASE expression that does *not* include the optional case operand.

Example

Suppose a company is considering giving raises to *all* of its employees, with a different percentage for each employee level:

- level-1 employees get a 5% raise
- level-2 employees get a l0% raise
- level-3 employees get a 15% raise.

First, you create the temporary table *Work.Payrollmaster3*, which is a copy of *Sasuser.payrollmaster*, the table containing

the employee salary data. The first 10 rows of *Work.Payrollmaster3* are shown below.

DateOfBirth	DateOfHire	EmpID	Gender	JobCode	Salary
16SEP1958	07JUN1985	1919	M	TA2	$48,126
19OCT1962	12AUG1988	1653	F	ME2	$49,151
08NOV1965	19OCT1988	1400	M	ME1	$41,677
04SEP1963	01AUG1988	1350	F	FA3	$46,040
16DEC1948	21NOV1983	1401	M	TA3	$54,351
29APR1952	11JUN1978	1499	M	ME3	$60,235
09JUN1960	04OCT1988	1101	M	SCP	$26,212
03APR1959	14FEB1979	1333	M	PT2	$124,048
20JAN1961	05DEC1988	1402	M	TA2	$45,661
26DEC1966	09OCT1987	1479	F	TA3	$54,299

Next, you create a PROC SQL step that updates rows by using an UPDATE statement that contains a SET clause and a CASE expression:

```
proc sql;
   update work.payrollmaster3
      set salary=salary*
         case
            when substr(jobcode,3,1)='1'
                 then 1.05
            when substr(jobcode,3,1)='2'
                 then 1.10
            when substr(jobcode,3,1)='3'
                 then 1.15
            else 1.08
         end;
```

In this example, the CASE expression contains three WHEN clauses, one for each subset of rows (level-1, level-2, and level-3 employees), followed by an ELSE clause to handle any rows that do not meet the expected conditions.

The first 10 rows of *Work.Payrollmaster3*, after the rows have been updated, are shown below.

DateOfBirth	DateOfHire	EmpID	Gender	JobCode	Salary
16SEP1958	07JUN1985	1919	M	TA2	$52,939
19OCT1952	12AUG1988	1653	F	ME2	$54,066
08NOV1965	19OCT1988	1400	M	ME1	$43,761
04SEF1963	01AUG1988	1350	F	FA3	$52,946
16DEC1948	21NOV1983	1401	M	TA3	$62,504
29APR1952	11JUN1978	1499	M	ME3	$69,270
09JUN1960	04OCT1988	1101	M	SCP	$28,309
03APR1959	14FEB1979	1333	M	PT2	$136,453
20JAN1961	05DEC1988	1402	M	TA2	$50,227
26DEC1966	09OCT1987	1479	F	TA3	$62,444

By comparing the values of **`Salary`** in the original and updated versions of *Work.Payrollmaster3* (as shown above), you can see how the values have changed according to the job level indicated in the **`JobCode`**.

Updating Rows by Using the CASE Expression with a Case Operand

If the expression in the SET clause uses an equals (=) comparison operator, you might use the optional case operand in the CASE expression. Consider PROC SQL step that was shown in the preceding example, and see how the CASE expression in the UPDATE statement can be rewritten by using the alternate syntax.

Example

In the following PROC SQL step, which was shown earlier, the CASE expression contains three WHEN-THEN clauses. These clauses contain similar expressions, each of which specifies the same SUBSTR function:

```
proc sql;
   update work.payrollmaster_new2
      set salary=salary*
         case
            when substr(jobcode,3,1)='1'
                  then 1.05
            when substr(jobcode,3,1)='2'
                  then 1.10
            when substr(jobcode,3,1)='3'
                  then 1.15
            else 1.08
         end;
```

Because the expression in this SET clause uses an equals (=) operator, you can restructure the CASE expression for more efficient processing. In the alternate syntax, the repeated SUBSTR function is removed from each WHEN-THEN clause and is placed after the keyword CASE, as an operand:

```
proc sql;
   update work.payrollmaster_new2
      set salary=salary*
         case substr(jobcode,3,1)
            when '1'
                 then 1.05
            when '2'
                 then 1.10
            when '3'
                 then 1.15
            else 1.08
         end;
```

Using the alternate syntax, the SUBSTR function is evaluated only once, so this PROC SQL step is more efficient than the original version.

Note You might use the case operand syntax *only* if the SET clause expression uses the equals (=) comparison operator.

Using the CASE Expression in the SELECT Statement

You can use the CASE expression in three different PROC SQL statements: UPDATE, INSERT, and SELECT. In the SELECT statement, you can use the CASE expression within a new column definition to specify different values for different subsets of rows.

Example

Suppose you want to generate an output report that displays employee names, job codes, and job levels. Your PROC SQL query selects **`LastName`** and **`FirstName`** from *Sasuser.Staffmaster*, and **`JobCode`** from *Sasuser.payrollmaster*. The SELECT statement must define **`JobLevel`** as a new column, because it does not exist as a separate column in either table.

You want to assign the values of **`JobLevel`**, based on the number at the end of each jobcode. (The number at the end of each **`JobCode`** value is expected to be *1, 2*, or *3*.) To create **`JobLevel`**, you can use the case operand form of the CASE expression to specify the three possible conditions (plus an ELSE condition, just in case).

The PROC SQL query is shown below:

```
proc sql outobs=10;
   select lastname, firstname, jobcode,
          case substr(jobcode,3,1)
             when '1'
                 then 'junior'
             when '2'
```

```
            then 'intermediate'
         when '3'
            then 'senior'
         else 'none'
      end as JobLevel
from sasuser.payrollmaster,
     sasuser.staffmaster
where staffmaster.empid=
      payrollmaster.empid;
```

LastName	FirstName	JobCode	JobLevel
ADAMS	GERALD	TA2	Intermediate
ALEXANDER	SUSAN	ME2	Intermediate
APPLE	TROY	ME1	junior
ARTHUR	BARBARA	FA3	senior
AVERY	JERRY	TA3	senior
BAREFOOT	JOSEPH	ME3	senior
BAUCOM	WALTER	SCP	none
BLAIR	JUSTIN	PT2	Intermediate
BLALOCK	RALPH	TA2	Intermediate
BOSTIC	MARIE	TA3	senior

The SELECT clause uses the CASE expression to assign a value of *junior, intermediate, senior*, or *none* to each row in the new **`JobLevel`** column.

Deleting Rows in a Table

Overview

To delete some or all of the rows in a table, use the *DELETE statement*. When the statement is successfully executed, the SAS log shows a message that indicates the number of rows that have been deleted.

General form, DELETE statement for deleting rows in a table:

```
DELETE FROM table-name
     <WHEREexpression>;
```

where

table-name

specifies the name of the table in which rows will be deleted.

WHERE

is optionally used to specify an *expression* that subsets the rows to be deleted.

Caution If you want to delete only a subset of rows in the table, you must specify a WHERE clause or *all* rows in the table will be deleted.

Note You can also use the DELETE statement to delete rows in a table that underlies a PROC SQL view. For more information about referencing a PROC SQL view in a DELETE statement, see "Creating and Managing Views Using PROC SQL" on page 260.

Example

Suppose you want to delete the records for all frequent-flyer program members who have used up all of their frequent flyer

miles or have spent more miles than they had in their accounts.

First, you create the temporary table *Work.Frequentflyers2* by copying a subset of columns from the existing table *Sasuser.Frequentflyers:*

```
proc sql;
   create table work.frequentflyers2 as
      select ffid, milestraveled,
             pointsearned, pointsused
        from sasuser.frequentflyers;
```

The first 10 rows of *Work.Frequentflyers2* are shown below.

FFID	MilesTraveled	PointsEarned	PointsUsed
WD7152	30833	31333	0
WD8472	25570	26070	0
WD1576	56144	58644	27000
WD3947	40922	45922	23000
WD9347	4839	9839	0
WD8375	30007	30507	25000
WD7208	43943	49443	30000
WD6061	60142	60642	40000
WD0646	87044	89544	25000
WD9829	1901	4401	0

Next, you write a PROC SQL step that deletes the specified rows:

```
proc sql;
   delete from work.frequentflyers2
      where pointsearned-pointsused <= 0;
```

A message in the SAS log tells you how many rows were deleted.

Table 5.17: SAS Log

```
NOTE: 13 rows were deleted from WORK.FREQUENTFLYERS2
```

Tip To delete *all* of the rows in the table, remove the WHERE clause from the DELETE statement.

Altering Columns in a Table

Overview

You have seen how to delete rows in a table using the DELETE statement. To add, drop (delete), or modify *columns* in a table, use the *ALTER TABLE statement*.

General form, ALTER TABLE statement:

```
ALTER TABLE table-name
     <ADDcolumn-deftnition-1<, … column-definition-n>>
     <DROPcolumn-name-1<,… column-name-n>>
     <MODlFYcolumn-definition-1<, … coluinn-definition-n>>',
```

where

table-name

specifies the name of the table in which columns will be added, dropped, or modified.

<ADD, DROP, MODIFY>

at least one of the following clauses must be specified:

ADD

specifies one or more *column-definitions* for columns to be added.

DROP

specifies one or more *column-names* for columns to be dropped (deleted).

MODIFY

specifies one or more *column-definitions* for columns to be modified, where *column-definition* specifies a column to be added or modified, and is formatted as follows:

column-name data-type <(column-width)> <column-modifier-1

<…column-modifier-n>>

In all three clauses, multiple *column-definitions* or *column-names* must be separated by commas.

Note You *cannot* use the ALTER TABLE statement with views.

Note The ALTER TABLE statement also supports similar clauses that add, drop, and modify integrity constraints in an existing table. These clauses are not discussed in this chapter. To find out more about adding, dropping, and modifying integrity constraints, see the SAS documentation for the SQL procedure.

Consider each type of modification that can be made to a column by using the ALTER TABLE statement.

Adding Columns to a Table

To add columns to a table, use the *ADD clause* in the *ALTER TABLE statement*. The ADD clause specifies one or more *column definitions*. The syntax of a column definition is the same as in the CREATE TABLE statement:

column-name data-type <(column-width)> <column-modifier-1<column-modifier-n>>

However, in the ALTER statement, the entire group of column definitions is *not* enclosed in parentheses.

Example

Suppose you are working with the temporary table *Work.Payrollmaster4*, which is an exact copy of the existing table *Sasuser.payrollmaster*. The first 10 rows of *Work.Payrollmaster4* are shown below.

DateOfBirth	DateOfHire	EmpID	Gender	JobCode	Salary
16SEP1958	07JUN1985	1919	M	TA2	$48,125
19OCT1962	12AUG1988	1653	F	ME2	$49,151
03NOV1965	19OCT1988	1400	M	ME1	$41,677
04SEP1963	01AUG1988	1350	F	FA3	$46,040
16DEC1948	21NOV1983	1401	M	TA3	$54,351
29APR1952	11JUN1978	1499	M	ME3	$60,235
09JUN1960	04OCT1988	1101	M	SCP	$26,212
03APR1959	14FEB1979	1333	M	PT2	$124,048
20JAN1961	05DEC1988	1402	M	TA2	$45,661
26DEC1966	09OCT1987	1479	F	TA3	$54,299

The following PROC SQL step uses the ADD clause in the ALTER TABLE statement to add the columns **Bonus** and **Level** to *Work.Payrollmaster4:*

```
proc sql;
   alter table work.payrollmaster4
```

```
    add Bonus num format=comma10.2,
        Level char(3);
```

The first 10 rows of *Work.Payrollmaster4*, with the two new columns added, are shown below.

DateOfBirth	DateOfHire	EmpID	Gender	JobCode	Salary	Bonus	Level
16SEP1958	07JUN1985	1919	M	TA2	$48,126	.	
19OCT1962	12AUG1988	1853	F	ME2	$49,151	.	
08NOV1965	19OCT1988	1400	M	ME1	$41,677	.	
04SEP1963	01AUG1988	1350	F	FA3	$46,040	.	
16DEC1948	21NOV1983	1401	M	TA3	$54,351	.	
29APR1952	11JUN1978	1499	M	ME3	$60,235	.	
09JUN1960	04OCT1988	1101	M	SCP	$26,212	.	
03APR1959	14FEB1973	1333	M	PT2	$124,048	.	
20JAN1961	05DEC1988	1402	M	TA2	$45,661	.	
26DEC1966	09OCT1987	1479	F	TA3	$54,299	.	

Use the UPDATE statement to populate the new columns.

Dropping Columns from a Table

To drop (delete) existing columns from a table, use the *DROP clause* in the *ALTER TABLE statement*. The DROP clause specifies one or more column names, and multiple column names are separated by commas.

Example

Suppose you want to drop the existing columns **Bonus** and **Level** from the temporary table *Work.Payrollmaster4*. (These two columns were added to the table in the example in the previous section.) The first 10 rows of *Work.Payrollmaster4* are shown below.

DateOfBirth	DateOfHire	EmpID	Gender	JobCode	Salary	Bonus	Level
16SEP1958	07JUN1985	1919	M	TA2	$48,126	.	
19OCT1962	12AUG1988	1653	F	ME2	$49,151	.	
08NOV1965	19OCT1988	1400	M	ME1	$41,677	.	
04SEP1963	01AUG1988	1350	F	FA3	$46,040	.	
16DEC1948	21NOV1983	1401	M	TA3	$54,351	.	
29APR1952	11JUN1978	1499	M	ME3	$60,235	.	
09JUN1960	04OCT1988	1101	M	SCP	$26,212	.	
03APR1959	14FEB1979	1333	M	FT2	$124,048	.	
20JAN1961	05DEC1988	1402	M	TA2	$45,661	.	
26DEC1966	09OCT1987	1479	F	TA3	$54,299	.	

The following PROC SQL step uses the DROP clause in the ALTER TABLE statement to drop the columns **Bonus** and **Level** from *Work.Payrollmaster4:*

```
proc sql;
   alter table work.payrollmaster4
     drop bonus, level;
```

The first 10 rows of *Work.Payrollmaster4*, with **Bonus** and **Level** deleted, are shown below.

DateOfBirth	DateOfHire	EmpID	Gender	JobCode	Salary
16SEP1958	07JUN1985	1919	M	TA2	$48,125

19OCT1962	12AUG1988	1653	F	ME2	$49,151
03NOV1965	19OCT1988	1400	M	ME1	$41,677
04SEP1963	01AUG1988	1350	F	FA3	$46,040
15DEC1948	21NOV1983	1401	M	TA3	$54,351
29APR1952	11JUN1978	1499	M	ME3	$60,235
09JUN1960	04OCT1988	1101	M	SCP	$26,212
03APR1959	14FEB1979	1333	M	PT2	$124,048
20JAN1961	05DEC1988	1402	M	TA2	$45,661
26DEC1966	09GCT1987	1479	F	TA3	$54,299

Modifying Columns in a Table

To modify the attributes of one or more existing columns in a table, use the *MODIFY clause* in the *ALTER TABLE statement*. You can use the MODIFY clause to change a column's

- length (column width) — for a character column only
- informat
- format
- label.

Note You *cannot* use the MODIFY clause to

- change a character column to numeric or vice versa. To change a column's data type, drop the column and then add it (and its data) again, or use the DATA step.
- change a column's name. You cannot change this attribute by using the ALTER TABLE statement. Instead, you can use the SAS data set option RENAME= or the DATASETS procedure with the RENAME statement. You can find out more about the DATASETS procedure with the RENAME statement in "Creating Samples and Indexes" on page 470.

Like the ADD clause, the MODIFY clause specifies one or more *column definitions*, each of which consists of

column-name <data-type (column-width)> <column-modifier-1<column-modifier-n>>

In each column definition, you specify the required element (the column name), followed by any of the optional attributes that you want to modify.

Note When you use a column definition to add a new column by using the ADD clause in the ALTER TABLE statement, or to specify a new column in the CREATE TABLE statement, *data-type* is a required element. However, when you are using a column definition in the MODIFY clause in the ALTER TABLE statement, as shown in the following example, *data-type* is never required for numeric columns and is optional for character columns. You must specify *data-type {column-width)* only if you want to modify the column width of a character column.

Note When modifying the width of a character variable it is possible to truncate the variable's value if the length specification is too small.

```
alter table work.payrollmaster
modify jobcode char(2);
select * from payrollmaster;
```

Example

Suppose you want to modify the attributes of the existing column **`Salary`** in the temporary table *Work.Payrollmaster4*. The first 10 rows of *Work.Payrollmaster4* (as it existed at the end of the previous example) are shown below.

DateOfBirth	DateOfHire	EmpID	Gender	JobCode	Salary
16SEP1958	07JUN1985	1919	M	TA2	$48,125

19OCT1962	12AUG1988	1653	F	ME2	$49,151
03NQV1965	19OCT1988	1400	M	ME1	$41,677
04SEP1963	01AUG1988	1350	F	FA3	$46,040
15DEC1948	21NOV1983	1401	M	TA3	$54,351
29APR1952	11JUN1978	1499	M	ME3	$60,235
09JUN1960	04OCT1988	1101	M	SCP	$26,212
0GAPR1959	14FEB1979	1333	M	PT2	$124,048
20JAN1961	05DEC1988	1402	M	TA2	$45,661
26DEC1966	09OCT1987	1479	F	TA3	$54,299

The column **Salary** isa numeric field that currently has the format DOLLAR9. The following PROC SQL step modifies the format and adds a label for **Salary:**

```
proc sql;
   alter table work.payrollmaster4
      modify salary format=dollar11.2 label="Salary Amt";
```

The first 10 rows of *Work.Payrollmaster4*, with the new column attributes for **Salary**, are shown below.

DateOfBirth	DateOfHire	EmpID	Gender	JobCode	Salary Amt
15SEP1958	07JUN1985	1919	M	TA2	$48,126.00
19OCT1952	12AUG1988	1653	F	ME2	$49,151.00
08NOV1965	19OCT1988	1400	M	ME1	$41,677.00
04SEP1963	01AUG1988	1350	F	FA3	$46,040.00
15DEC1948	21NOV1983	1401	M	TA3	$54,351.00
29APR1952	11JUN1978	1499	M	ME3	$60,235.00
09JUN1960	04OCT1988	1101	M	SCP	$26,212.00
03APR1959	14FEB1979	1333	M	PT2	$124,048.00
20JAN1961	05DEC1988	1402	M	TA2	$45,561.00
26DEC1966	09OCT1987	1479	F	TA3	$54,299.00

Adding, Dropping, and Modifying Columns in a Single Statement

In the last few examples, the ALTER TABLE statement has made only one alteration to columns in a table, by using just one clause. However, you can include multiple clauses in a single ALTER TABLE statement to add, drop, and modify columns all at once.

Example

Suppose you want to use a single ALTER TABLE statement to make *all* of the following alterations to the table *Work.Payrollmaster4:*

- add the new column **Age**, by using the ADD clause
- change the format of the **DateOfHire** column (which is currently DATE9.) to MMDDYY10., by using the MODIFY clause
- drop the **DateOfBirth** and **Gender** columns, by using the DROP clause.

The first 10 rows of *Work.Payrollmaster4*, as it was at the end of the last example, are shown below.

DateOfBirth	DateOfHire	EmpID	Gender	JobCode	Salary Amt
16SEP1958	07JUN1985	1919	M	TA2	$48,126.00
19OCT1962	12AUG1988	1653	F	ME2	$49,151.00

03NOV1965	19OCT1988	1400	M	M E1	$41,677.00
04SEP1963	01AUG1988	1350	F	FA3	$46,040.00
16DEC1948	21NOV1983	1401	M	TA3	$54,351.00
29APR1952	11JUN1978	1499	M	ME3	$60,235.00
09JUN1960	04OCT1988	1101	M	SCP	$26,212.00
03APR1959	14FEB1979	1333	M	PT2	$124,048.00
20JAN1961	05DEC1988	1402	M	TA2	$45,661.00
26DEC1966	09OCT1987	1479	F	TA3	$54,299.00

The following PROC SQL step uses multiple clauses in the ALTER TABLE statement to make all three of the alterations listed above:

```
proc sql;
   alter table work.payrollmaster4
      add Age num
      modify dateofhire date format=mmddyy10.
      drop dateofbirth, gender;
```

The first 10 rows of *Work.Payrollmaster4*, with the three alterations, are shown below.

DateOfHire	EmpID	JobCode	Salary Amt	Age
06/07/1985	1919	TA2	$48,126.00	.
08/12/1988	1653	ME2	$49,151.00	.
10/19/1988	1400	ME1	$41,677.00	.
08/01/1988	1350	FA3	$46,040.00	.
11/21/1983	1401	TA3	$54,351.00	.
06/11/1978	1499	ME3	$60,235.00	.
10/04/1988	1101	SCP	$26,212.00	.
02/14/1979	1333	PT2	$124,048.00	.
12/05/1988	1402	TA2	$45,661.00	.
10/09/1987	1479	TA3	$54,299.00	.

Use the UPDATE statement to populate the new columns.

Dropping Tables

Overview

To drop (delete) one or more entire tables, use the *DROP TABLE statement*.

General form, DROP TABLE statement:

DROP TABLE *table-name-1 <, …table-name-n>;*

where

table-name

specifies the name of a table to be dropped, and can be one of the following:

- a one-level name
- a two-level libref.table name
- a physical pathname that is enclosed in single quotation marks.

Example

In the last few examples, you made several alterations to the temporary table *Work.Payrollmaster4*. Now you decide that you do not need this table anymore. The following PROC SQL step uses the DROP TABLE statement to drop *Work.Payrollmaster4:*

```
proc sql;
   drop table work.payrollmaster4;
```

The SAS log displays a message indicating that the table has been dropped:

Table 5.18: SAS Log

```
NOTE: Table WORK.PAYROLLMASTER4 has been dropped.
```

Summary

Contents

This section contains the following topics.

Text Summary

Understanding Methods of Creating Tables

You can use the CREATE TABLE statement to create a table in three different ways:

- create a table with no rows (an empty table) by defining columns
- create an empty table that is like another table
- create a table that contains rows, based on a query result.

Creating an Empty Table by Defining Columns

You can create a table with no rows by using a CREATE TABLE statement that contains column specifications. A column specification includes the following elements: column name (required), data type (required), column width (optional), one or more column modifiers (optional), and a column constraint (optional).

Displaying the Structure of a Table

To display, in the SAS log, a list of a table's columns and their attributes and other information about a table, use the DESCRIBE TABLE statement.

Creating an Empty Table That Is like AnotherTable

To create a table with no rows that has the same structure as an existing table, use a CREATE TABLE statement that contains the keyword LIKE. To specify a subset of columns to be copied from the existing table, use the SAS data set options DROP= or KEEP= in your CREATE TABLE statement.

Creating a Table from a Query Result

To create a new table that contains both columns and rows that are derived from an existing table or set of tables, use a CREATE TABLE statement that includes the keyword AS and the clauses that are used in a query. This method enables you to copy an existing table quickly.

Inserting Rows of Data into a Table

The INSERT statement can be used in three ways to insert rows of data in existing tables, either empty or populated. You can insert rows by using

- the SET clause to specify column names and values in pairs
- the VALUES clause to specify a list of values
- the clauses that are used in a query to return rows from an existing table.

Creating a Table That Has Integrity Constraints

Integrity constraints are rules that you can specify in order to restrict the data values that can be stored for a column in a table. To create a table that has integrity constraints, use a CREATE TABLE statement. Integrity constraints can be defined in two different ways in the CREATE TABLE statement:

- by specifying a column constraint in a column specification
- by using a constraint specification.

Handling Errors in Row Insertions

When you add rows to a table that has integrity constraints, PROC SQL evaluates the new data to ensure that it meets the conditions that are determined by the integrity constraints. When you use the INSERT or UPDATE statement to add or modify data in a table, you can use the UNDO_POLICY= option in the PROC SQL statement to specify whether PROC SQL will make or undo the changes you submitted up to the point of the error.

Displaying Integrity Constraints for a Table

To display the integrity constraints for a specified table in the SAS log, use the DESCRIBE TABLE CONSTRAINTS statement.

Updating Values in Existing Table Rows

To modify data values in some or all of the existing rows in a table, use the UPDATE statement with

- a SET clause and (optionally) a WHERE clause that specifies a single expression to update rows. To update rows with multiple expressions, use multiple UPDATE statements.
- a CASE expression that specifies multiple expressions to update rows. The CASE expression can be specified without an optional case operand or, if the expression in the SET clause uses an equals (=) comparison operator, with a case operand.

The CASE expression can also be used in the SELECT statement in a new column definition to specify different values for different subsets of rows.

Deleting Rows in a Table

To delete some or all of the rows in a table, use the DELETE statement.

Altering Columns in a Table

To alter columns in a table, use the ALTER TABLE statement that contains one or more of the following clauses:

- the ADD clause, to add one or more columns to a table
- the DROP clause, to drop (delete) one or more columns in a table
- the MODIFY clause, to modify the attributes of columns in a table.

Dropping Tables

To drop (delete) one or more entire tables, use the DROP TABLE statement.

Syntax

```
PROC SQL <UNDO_POLICY=REQUIRED| NONE | OPTIONAL>;
  CREATE TABLE table-name
```

```
    (column-specification-1><,
    …column-specification-n><,
    constraint-specification-1><,
    …constraint-specification-n>);
  CREATE TABLE table-name
    (column-definition <column-constraint-1<,… column-constraint-n>>,
        column-definition <column-constraint-1<,… column-constraint-n>>);
  CREATE TABLE table-name
         (DROP | KEEP =column-1<…,column-n>)
    LIKE table-1;
  CREATE TABLE table-name AS
      SELECT column-1<, … column-n>
          FROM table-1 | view-1<,… table-n | view-n>
          <optional queiy clauses>;
  DESCRIBE TABLE table-name-1<, … table-name-n>;
  DESCRIBE TABLE CONSTRAINTS table-name-1<, … table-name-n>;
  INSERT INTO table-name<(target-column-1<, … target-column-n)>
       SET column-1= value-1<, … column-n=value-n>
       <… SET column-1-value-1<,… column-n=value-n>>',
  INSERT INTO table-name<(target-column-1<, … target-column-n)>
       VALUES(value-1<,… value-n>)
       <… VALUES(value-1<, … value-n>)>;
  INSERT INTO table-name<(target-column-1<, … target-column-n)>
       SELECT column-1<, … column-n>
           FROM table-1 | view-1<,… table-n | view-n>
           <optional query clauses>;
  UPDATE table-name
      SET column-1=expression<, … column-n= expression>>
            <WHEREexpression>;

  UPDATE table-name
      SET column-1 expression<, … column-n expression>>
             CASE<case-operand>
               WHEN when-condition THEN result-expression
               <…WHENwhen-conditionTHENresult-expression>
               <ELSEresidt-expression>
             END;

  DELETE FROM table-name
      <WHEREexpression>;
  ALTER TABLE table-name
      <ADDcolumn-definition-1<, … column-definition-n>>
      <DROPcolumn-name-1<,… column-name-n>>
      <MODIFYcolumn-definition-1<, … column-defmition-n>>;
  DROP TABLE table-name-1<, … table-name-n>;
QUIT;
```

Sample Programs

Creating an Empty Table by Defining Columns

```
proc sql;
   create table work.discount
          (Destination char(3),
          BeginDate num Format=date9.,
          EndDate num format=date9.,
          Discount num);
quit;
```

Creating an Empty Table That Is like AnotherTable

```
proc sql;
   create table work.flightdelays2
          (drop=delaycategory destinationtype)
      like sasuser.flightdelays;
quit;
```

Creating a Table from a Query Result

```
proc sql;
   create table work.ticketagents as
      select lastname, firstname,
             jobcode, salary
         from sasuser.payrollmaster,
              sasuser.staffmaster
         where payrollmaster.empid
               = staffmaster.empid
               and jobcode contains 'TA';
quit;
```

Displaying the Structure of a Table

```
proc sql;
   describe table work.discount;
quit;
```

Inserting Rows into a Table by Specifying Column Names and Values

```
proc sql;
   insert into work.discount
        set destination='LHR',
            begindate='01MAR2000'd,
            enddate='05MAR2000'd,
            discount=.33
        set destination='CPH',
            begindate='03MAR2000'd,
            enddate='10MAR2000'd,
            discount=.15;
quit;
```

Inserting Rows into a Table by Specifying Lists of Values

```
proc sql;
   insert into work.discount (destination,
          begindate,enddate, discount)
      values ('LHR','01MAR2000'd,
              '05MAR2000'd,.33)
      values ('CPH','03MAR2000'd,
              '10MAR2000'd,.15);
quit;
```

Inserting Rows into a Table from a Query Result

```
proc sql;
   insert into work.payrollchanges2
      select empid, salary, dateofhire
         from sasuser.payrollmaster
         where empid in ('1919','1350','1401');
quit;
```

Creating a Table That Has Integrity Constraints

```
proc sql;
   create table work.employees
          (Name char(10),
          Gender char(1),
          HDate date label='Hire Date' not null,
          constraint prim_key primary key(name),
          constraint gender check(gender in ('M' 'F')));
quit;
```

Displaying Integrity Constraints for a Table

```
proc sql;
   describe table constraints work.discount4;
quit;
```

Updating Rows in a Table Based on an Expression

```
proc sql;
   update work.payrollmaster_new
      set salary=salary*1.05
      where jobcode like '__1';
quit;
```

Updating Rows in a Table by Using a CASE Expression

```
proc sql;
   update work.payrollmaster_new
      set salary=salary*
         case
            when substr(jobcode,3,1)='1'
                 then 1.05
            when substr(jobcode,3,1)='2'
                 then 1.10
            when substr(jobcode,3,1)='3'
                 then 1.15
            else 1.08
         end;
quit;
```

Updating Rows in a Table by Using a CASE Expression (Alternate Syntax)

```
proc sql outobs=10;
   select lastname, firstname, jobcode,
          case substr(jobcode,3,1)
               when '1'
                    then 'junior'
               when '2'
                    then 'intermediate'
               when '3'
                    then 'senior'
               else 'none'
          end as JobLevel
      from sasuser.payrollmaster,
           sasuser.staffmaster
      where staffmaster.empid=
            payrollmaster.empid;
quit;
```

Deleting Rows in a Table

```
proc sql;
   delete from work.frequentflyers2
      where pointsearned-pointsused<=0;
quit;
```

Adding, Modifying, and Dropping Columns in a Table

```
proc sql;
   alter table work.payrollmaster4
      add Age num
      modify dateofhire date format=mmddyy10.
      drop dateofbirth, gender;
quit;
```

Dropping a Table

```
proc sql;
   drop table work.payrollmaster4;
quit;
```

Points to Remember

- The CREATE TABLE statement generates only a table as output, not a report.
- The UPDATE statement does not insert new rows into a table. To insert rows, you must use the INSERT statement.

Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1. Which of the following PROC SQL steps creates a new table by copying only the column structure (but not the rows) of an existing table? ?

a.
```
a. proc sql;
     create table work.newpayroll as
        select *
           from sasuser.payrollmaster;
```

b.
```
b.  proc sql;
     create table work.newpayroll
        like sasuser.payrollmaster;
```

c.
```
c.  proc sql;
     create table work.newpayroll
        copy sasuser.payrollmaster;
```

d.
```
d.  proc sql;
     create table work.newpayroll
        describe sasuser.payrollmaster;
```

2. Which of the following PROC SQL steps creates a table that contains rows for the level-1 flight attendants only? ?

a.
```
a.  proc sql;
     create table work.newpayroll as
        select *
           from sasuser.payrollmaster
           where jobcode='FA1';
```

b.
```
b.  proc sql;
     create work.newpayroll as
        select *
           from sasuser.payrollmaster
           where jobcode='FA1';
```

c.
```
c.  proc sql;
     create table work.newpayroll
        copy sasuser.payrollmaster
           where jobcode='FA1';
```

d.
```
d.  proc sql;
     create table work.newpayroll as
           sasuser.payrollmaster
        where jobcode='FA1';
```

3. Which of the following statements is true regarding the UNDO_POLICY=REQUIRED option? ?

a. It must be used with the REQUIRED integrity constraint.

b. It ignores the specified integrity constraints if any of the rows that you want to insert or update do not meet the constraint criteria.

c. It restores your table to its original state if any of the rows that you try to insert or update do not meet the specified integrity constraint criteria.

d. It allows rows that meet the specified integrity constraint criteria to be inserted or updated, but rejects rows that do not meet the integrity constraint criteria.

4. Which of the following is *not* a type of integrity constraint? ?

a. CHECK

b. NULL

c. UNIQUE

d. PRIMARY KEY

5. Which of the following PROC SQL steps deletes rows for all frequent-flyer program members who traveled less than 10,000 miles? ?

a.
```
a.  proc sql;
     delete rows
        from work.frequentflyers
        where milestraveled < 10000;
```

b.
```
b.  proc sql;
     drop rows
        from work.frequentflyers
        where milestraveled < 10000;
```

c.
```
c.  proc sql;
     drop table
        from work.frequentflyers
        where milestraveled < 10000;
```

d.
```
d.  proc sql;
     delete
        from work.frequentflyers
        where milestraveled < 10000;
```

6. Which of the following PROC SQL steps gives bonuses (in points) to frequent-flyer program members as follows: ?

- a 50% bonus for members who traveled less than 10,000 miles
- a 100% bonus for members who traveled 10,000 miles or more?

a.
```
a.  proc sql;
     update work.frequentflyers
     set pointsearned=pointsearned*
        case if milestraveled < 10000
                then 1.5
             if milestraveled >= 10000
                then 2
             else 1
        end;
```

b.
```
b.  proc sql;
     update work.frequentflyers
     set pointsearned=pointsearned*
        case when milestraveled < 10000
                  then 1.5
             when milestraveled >= 10000
                  then 2
             else 1
        end;
```

c.
```
c.  proc sql;
     update work.frequentflyers
     set pointsearned=pointsearned*
        case if milestraveled < 10000
```

```
                  then pointsearned*1.5
               if milestraveled >= 10000
                then pointsearned*2
             else 1
        end;
```

d.
```
d.  proc sql;
     update work.frequentflyers
     set pointsearned=pointsearned*
        case if milestraveled < 10000
                  then pointsearned*1.5
               if milestraveled >= 10000
                then pointsearned*2
             else pointsearned*1
      end;
```

7. Which of the following statements is used to add new rows to a table? ?

a. INSERT

b. LOAD

c. VALUES

d. CREATE TABLE

8. Which of the following statements regarding the ALTER TABLE statement is false? ?

a. It allows you to update column attributes.

b. It allows you to add new columns in your table.

c. It allows you to drop columns in your table.

d. It allows you to change a character column to a numeric column.

9. Which of the following displays the structure of a table in the SAS log? ?

a.
```
a.  proc sql;
     describe as
        select *
           from sasuser.payrollmaster;
```

b.
```
b.  proc sql;
     describe contents sasuser.payrollmaster;
```

c.
```
c.  proc sql;
     describe table sasuser.payrollmaster;
```

d.
```
d.  proc sql;
     describe * from sasuser.payrollmaster;
```

10. Which of the following creates an empty table that contains the two columns **`FullName`** and **`Age`**? ?

a.
```
a.  proc sql;
     create table work.names
            (FullName char(25), Age num);
```

b.
```
b.  proc sql;
     create table work.names as
            (FullName char(25), Age num);
```

c.
```
c.  proc sql;
     create work.names
            (FullName char(25), Age num);
```

d. d. proc sql;
create table work.names
set (FullName char(25), Age num);

Answers

1. Correct answer: b

   The CREATE TABLE statement that includes a LIKE clause copies the column names and attributes from an existing table into a new table. No rows of data are inserted.

2. Correct answer: a

   The CREATE TABLE statement that includes the AS keyword and query clauses creates a table and loads the results of the query into the new table. The WHERE clause selects only the rows for the level-1 flight attendants.

3. Correct answer: c

   UNDO POLICY=REQUIRED is the default setting for PROC SQL. This setting undoes *all* inserts or updates if 1 or more rows violate the integrity constraint criteria, and restores the table to its original state before the inserts or updates.

4. Correct answer: b

   The NOT NULL integrity constraint specifies that data is required and cannot have a null (missing) value.

5. Correct answer: d

   The DELETE statement deletes rows that are specified in the WHERE clause from the table. If no WHERE clause is specified, all rows are deleted. The DROP TABLE statement drops (deletes) an entire table; the syntax shown in option c is not valid.

6. Correct answer: b

   The UPDATE statement that includes a SET clause is used to modify rows in a table. WHEN-THEN clauses in the CASE expression enable you to update a column value based on specified criteria.

7. Correct answer: a

   The INSERT statement is used to insert new rows into a new or existing table. There is no LOAD statement in PROC SQL, VALUES is a clause, and the CREATE TABLE statement is used to create a table.

8. Correct answer: d

   The ALTER TABLE statement is used to modify attributes of existing columns (include the MODIFY clause), add new column definitions (include the ADD clause), or delete existing columns (include the DROP clause).

9. Correct answer: c

   The DESCRIBE TABLE statement lists the column attributes for a specified table.

10. Correct answer: a

The CREATE TABLE statement can include column specifications to create an empty table. The entire group of column specifications must be enclosed in a single set of parentheses. You must list each column's name, data type, and (for character columns) length. The length is specified as an integer in parentheses. Multiple column specifications must be separated by commas.